



DEV 430

Optimizing Software for 64-bit Windows® for Extended Systems...in 5 easy steps

Michael Wall

Senior Member of Technical Staff, AMD

Agenda

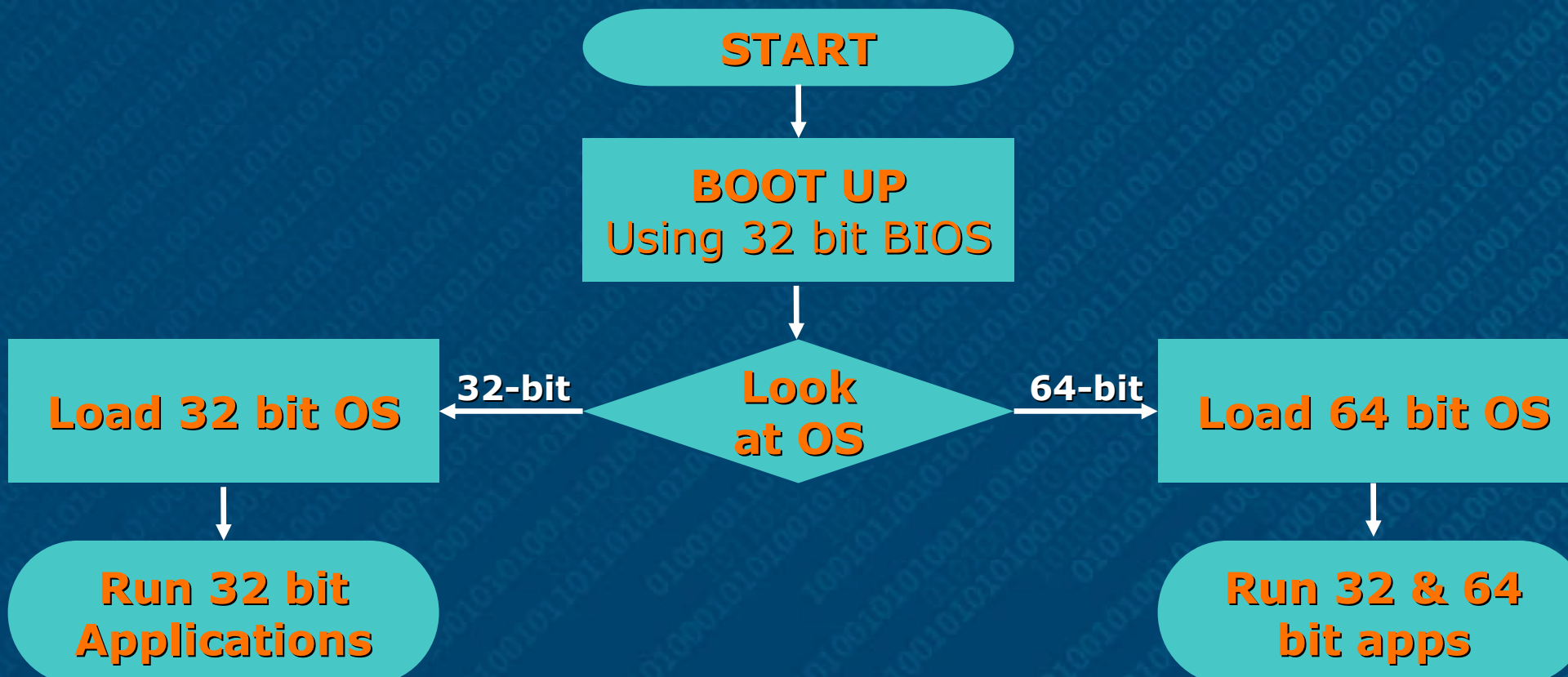
This talk is for Windows® developers who care about maximizing performance

- A very quick look at the 64-bit OS and compatibility
- A quick look at the AMD 64-bit processors
- Maximizing performance of 64-bit applications
 - Leveraging Large memory
 - **Tools, techniques, and extra registers: the 5 steps**
- Demo: 64-bit performance on a notebook
- Q & A, developer resources

64-bit Windows® for Extended Systems

32-bit and 64-bit on a single platform

An AMD64-based Processor can run both 32-bit and 64-bit Windows® operating systems

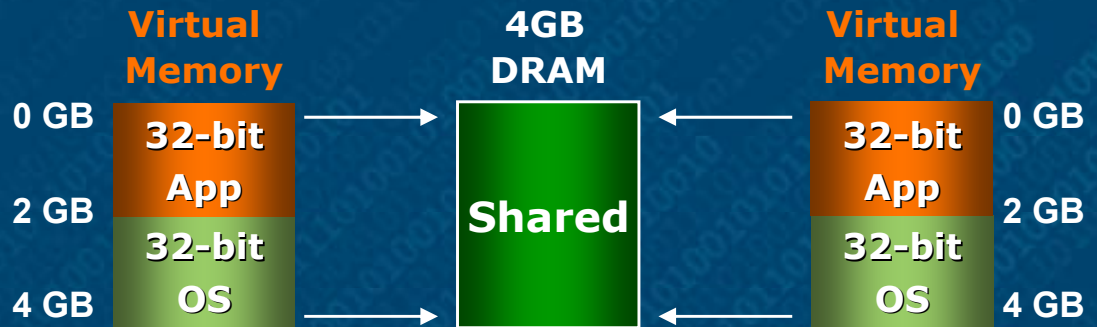


64-bit Windows® for Extended Systems

Why the 64-bit OS Benefits 32-bit Applications

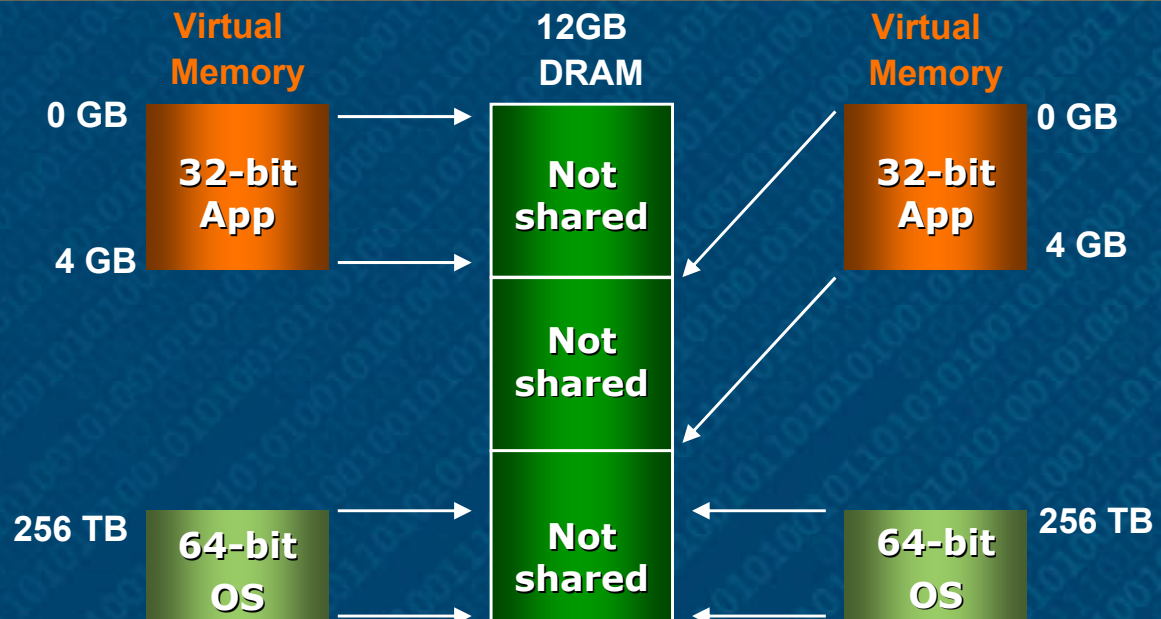
32-bit x86 system

- OS and applications share virtual and physical memory
- Results in a lot of paging of info in and out of memory
- Limits the size of files and datasets



64-bit Extended System

- OS uses Virtual Memory space outside range of 32-bits
- Application has exclusive use of virtual memory space
- OS can allocate each application dedicated portions of physical memory
- Reduces the amount of paging
- Supports larger file and dataset sizes



AMD64 Processor Family

AMD64 Technology

- **AMD Athlon™ 64**

Single processor
Desktop, Mobile



- **AMD Opteron™**

Multi-processor
Workstation, Server



AMD64 Processor Family

Windows® and AMD64 Technology

Unifying theme: Designed for Compatibility

- Processor: Native hardware support for 32-bit and 64-bit x86 code
- OS: 64-bit Windows® runs 32-bit and 64-bit applications side by side, seamlessly
- Source Code: A single C/C++ source code tree compiles to both 32-bit and 64-bit binaries

Porting and Optimizing for 64-bit

Two different ways to get more performance

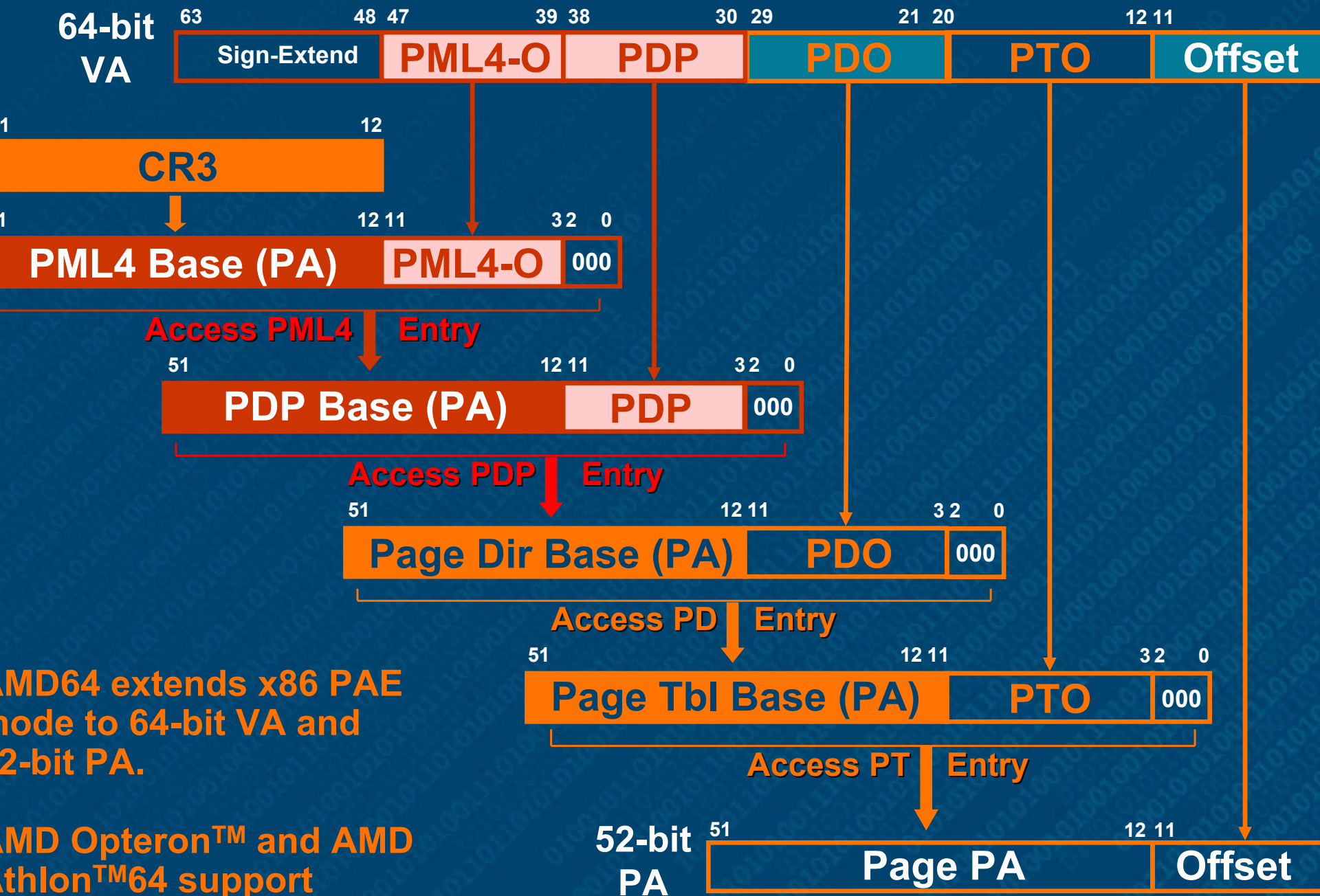
- Large Memory!!
 - Virtually unlimited address space
- Extra Registers in 64-bit mode
 - Twice as many General Purpose Registers (GPRs)
 - Twice as many SSE/SSE2 Registers

Large Memory and Performance

Memory is the obvious 64-bit advantage

- Instruction set implements a full 64-bit virtual address, in full 64-bit registers
- 52-bit physical address (4 million gigabytes)
- Current generation AMD CPUs support a 40-bit physical address (1 Terabyte)

AMD64 ISA 64-bit Virtual Addressing



AMD64 extends x86 PAE mode to 64-bit VA and 52-bit PA.

AMD Opteron™ and AMD Athlon™64 support 64-bit PA.

Large Memory and Performance

Certain applications naturally benefit

- Database
- Web server
- Terminal Server
- Application server
- Workstation: 3D animation, CAD, EDA

Large Memory and Performance

Creative use of big virtual address space

- Map files to memory:

CreateFileMapping and MapViewOfFile

- A single, flat address space for all data
- Let Windows manage disk and RAM
- Simplified programming model
- Performance scales up naturally with increased physical RAM

Port/Optimize Step 1: cleanup

Get your code “64-bit clean”

- Build your project using the /Wp64 compiler switch
 - This switch is supported by 32-bit VS.NET compiler!
- Warns about non-portable code
- Fix it so your project compiles cleanly to 32-bit target
- *Then* you're ready to use the 64-bit tools

Step 1: cleanup

- Clean-up example 1: use new “polymorphic” data types

- Original code stores a pointer in a LONG

```
LONG userdata = (LONG) &max_coordinate;
```

- When built for a 64-bit target, this will truncate the pointer (64-bit value) by storing in a LONG (32-bit size).

- Use LONG_PTR instead:

```
LONG_PTR userdata = (LONG_PTR) &max_coordinate;
```

- Data type LONG_PTR is “a long the size of a pointer” so it grows to 64 bits when you compile for 64-bit target (like `size_t`)

Step 1: cleanup

- Clean-up example 2: a few API calls have been updated

- Old API call uses a 32-bit LONG for user data

```
LONG udat = myUserData;  
  
LONG v = SetWindowLong( hWnd, GWL_USERDATA,  
    udat );
```

- New API call replaces the old one:

```
LONG_PTR udat = myUserData;  
  
LONG_PTR v = SetWindowLongPtr( hWnd,  
    GWLP_USERDATA, udat );
```

- The old call is deprecated; the new call works for 32-bit and 64-bit targets, so you can (and should) change all your code

Step 2: build

Get everything to build for 64-bit target

- There may be additional compiler warnings/errors
- Data structure alignment is a common trouble spot
 - Data is “naturally aligned” in structs
 - Pointer members and polymorphic members grow
 - Shared data (e.g. access by assembly code) needs special care
- Data files shared between 32-bit and 64-bit processes may need special handling, especially if they contain pointers
- Skip assembly code porting, for initial 64-bit build

Step 3: **measure and analyze!**

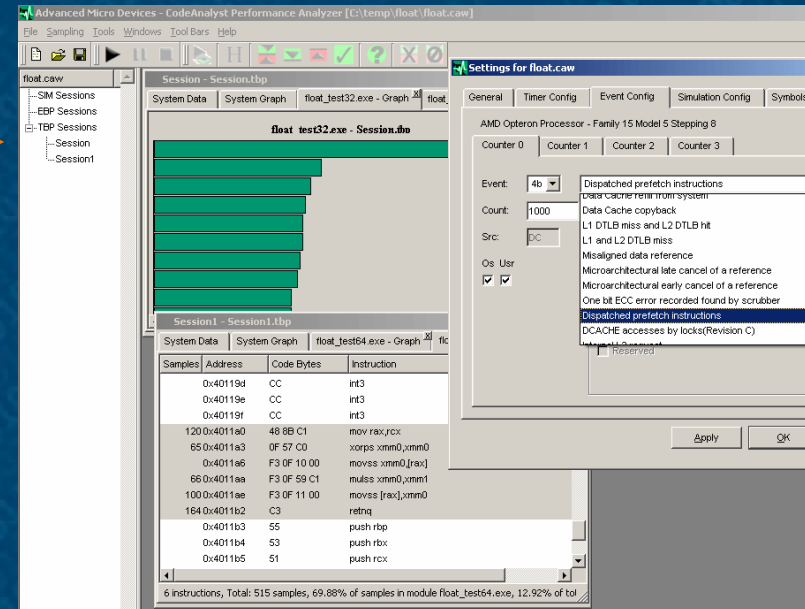
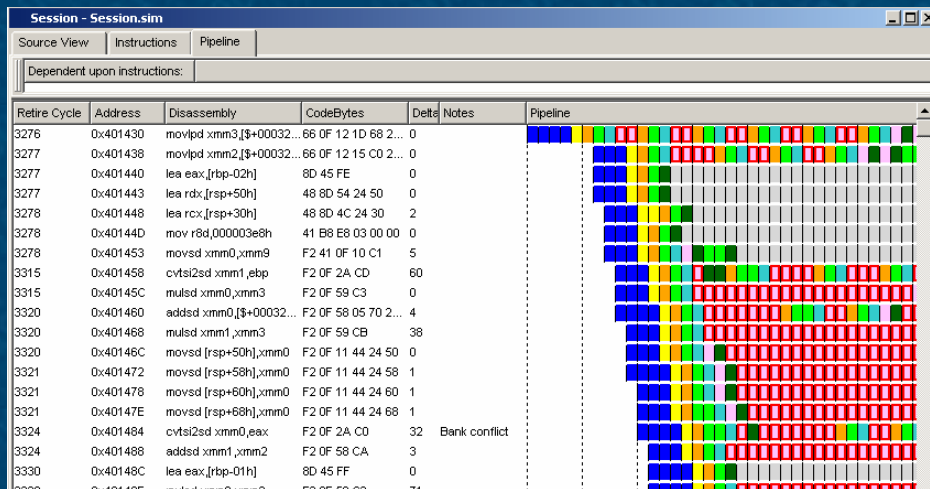
Benchmark and profile your 64-bit code

- Performance profile may differ from old 32-bit build
 - There are new drivers...
 - and there are new optimized libraries...
 - and a new 64-bit optimizing compiler...
 - all running on new CPU micro-architecture.
- **Focus optimization efforts on real, measured hot spots!**
- You may be surprised by where your code spends time

Step 3: measure and analyze

Use AMD CodeAnalyst to profile your 64-bit code!!

- ## ● Timer-based & event-based profiler



- You can get a detailed pipeline view of critical code sections

- CodeAnalyst can be downloaded at amd.com/developer section

Step 4: optimize

Tune for maximum 64-bit performance

- AMD64 technology offers 8 extra GPRs and 8 extra SSE registers... all used by the compiler
- Many old C/C++ tricks still work, some extra-well
 - The compiler optimizes, but you can help it at source level
- Compiler intrinsic functions for SSE, SSE2 and other funcs
 - Portable across 32-bit and 64-bit targets, compiler does reg allocation
- Assembly code can still be used for absolute max performance
 - Vectorize your 64-bit SSE code, tweak instruction scheduling, etc.

Step 4: optimize

- Use the 64-bit compiler's optimization features
- Good ol' standard optimization switches
 - Compile with `/O2` or `/O2b2`, and use `/fp:fast`
- Whole Program Optimization (WPO)
 - Compile with `/GL` and link with `/LTCG`
 - Enables more function inlining, other cross-module improvements
- Profile Guided Optimization (PGO or "Pogo")
 - Build instrumented binaries, run your workload, re-link
 - Can improve function layout (I-cache usage) and branch flow

Step 4: optimize

C/C++ optimization example 1: special loop unrolling for greater parallelism

- The compiler unrolls loops automatically, but manual unrolling can introduce *explicit parallelism*

```
double a[100], sum;  
sum = 0.0;  
for (int i = 0; i < 100; i++) {  
    sum += a[i];    // no parallelism possible  
}
```

- The compiler must use a single “sum” and perform addition in-order, creating a long dependency chain and leaving execution units idle much of the time
- How can this be improved?

Step 4: optimize

- Manually unroll the loop into *parallel dependency chains*

```
double a[100], sum, sum1, sum2, sum3, sum4;
sum1 = sum2 = sum3 = sum4 = 0.0;
for (int i = 0; i < 100; i += 4) {
    sum1 += a[i];           // these four
    sum2 += a[i+1];         // chains can
    sum3 += a[i+2];         // run in
    sum4 += a[i+3];         // parallel
}
sum = sum1 + sum2 + sum3 + sum4;
```

The switch
/fp:fast
can do this
automaticall
in many case

Use it!

- With 4 separate dependency chains the execution units can be kept busy with pipelined operations... over 3x faster here!
- This trick is particularly advantageous in more complex loops with AMD64 technology because of all the registers available.*

Step 4: optimize

C/C++ optimization example 2: aliasing

- Perhaps the biggest optimization roadblock for the compiler is *aliasing*. Pointers may step on each others' data

```
int *a, *b, *c;  
for (int i = 0; i < 100; i++) {  
    *a += *b++ - *c++; // b or c may point to a  
}
```

- The compiler must be cautious, and write to memory for each loop iteration. It cannot safely keep the sum “a” in a register.
- How can this be improved?

Step 4: optimize

- Use the `__restrict` keyword to help the compiler
- Apply your external knowledge that `*a` does not alias `*b` or `*c`

```
int* __restrict a;  
int *b, *c;  
for (int i = 0; i < 100; i++) {  
    *a += *b++ - *c++; // no aliases exist  
}
```

- Now the compiler can safely keep the sum in a register, and avoid many memory writes.
- Read more about keyword `__restrict`, `declspec(restrict)` and `declspec(noalias)` in Microsoft docs. They are powerful.

Step 4: optimize

C/C++ optimization example 3: struct data packing

- Structure data members are “naturally aligned”
- Padding may be added when you compile for 64-bit

```
struct foo_original { int a, void *b, int c };
```
- 12 bytes in 32-bit mode, but **24** bytes in 64-bit mode!
- Fix it by re-ordering elements for better packing

```
struct foo_new      { void *b, int a, int c };
```
- 12 bytes in 32-bit mode, only 16 bytes in 64-bit mode.
- Also re-order struct elements for better cache locality

Step 4: optimize

Compiler intrinsic examples: SSE and SSE2

```
__m128  __mm_mul_ss( __m128 a, __m128 b );
```

SSE MULSS scalar single-precision multiply instruction

```
__m128d  __mm_add_pd( __m128d a, __m128d b );
```

SSE2 ADDPD packed double-precision add instruction

```
__m128i  __mm_load_si128( __m128i *p );
```

SSE2 MOVDQA instruction for 128-bit integer

```
__m128d  __mm_set_pd( double x, double y );
```

SSE2 initialize a packed vector variable

```
__m128d  __mm_setzero_pd( );
```

SSE2 XORPD initialize packed double to zero

```
__m128i  __mm_cvtsi32_si128( int a );
```

SSE2 MOVD load a 32-bit int as lower bits of SSE2 reg

Step 4: optimize

compiler intrinsic examples: other goodies

```
void __cpuid( int* CPUInfo, int InfoType );
```

for detecting CPU features

```
unsigned __int64 __rdtsc( void );
```

for reading the timestamp counter (cycle counter) and
accurately measuring performance of critical code sections

```
__int64 __mul128( __int64 Multiplier, __int64 Multiplicand,  
                 __int64 *HighProduct );
```

fast multiply of two 64-bit integers, returning full 128-bit result
(lower 64 bits are return value, upper 64 bits by indirect pointer)

```
__int64 __mulh( __int64 a, __int64 b );
```

fast multiply of two 64-bit integers, returning the high 64 bits

```
void __mm_prefetch( char* p, int i );
```

software prefetch instruction for reading data into CPU cache

```
void __mm_stream_ps( float* p, __m128 a );
```

streaming store of data to memory, bypassing CPU cache

Step 4: optimize

C/C++ optimization example 4: cache control

- Data can be fetched with “non-temporal” tag

```
_mm_prefetch ( (char *) foo_ptr, _MM_HINT_NTA );
```

- Loads a 64-byte cache line into L1, won't disturb L2

- Streaming store: write directly to memory, not to cache

```
_mm_stream_ps ( (float *) foo_ptr, var_128 );
```

- var_128 is type `__m128` (really an SSE register)

- A bit awkward, pack 16 bytes and write to aligned address

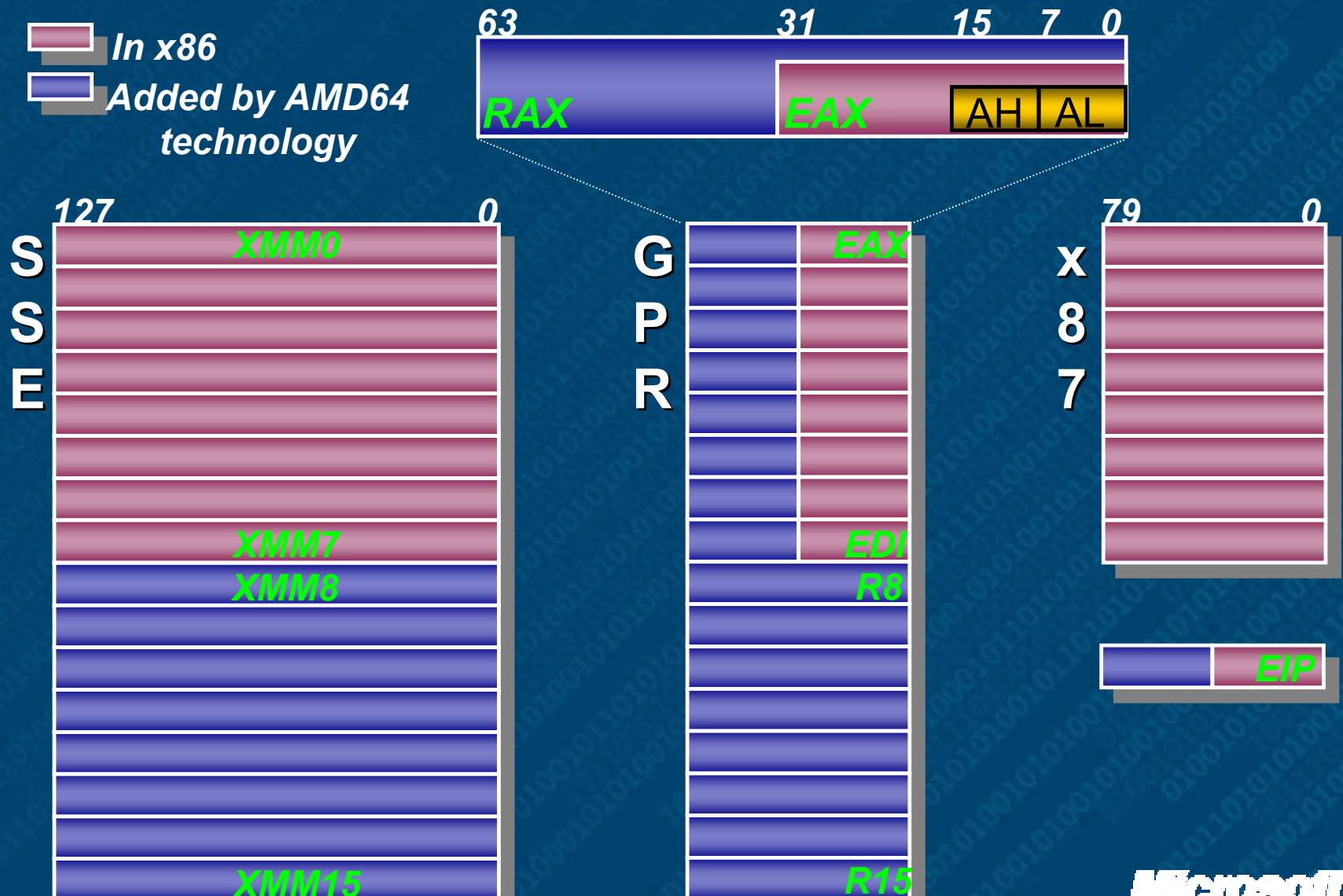
- Four 16-byte packets = 64-byte write-combine buffer

Step 4: optimize

- Assembly code is still worthwhile for maximum performance in certain critical inner loops
- In-line `_asm` code is not supported for 64-bit code use `MASM`
- Pay attention to prolog/epilog, it's different... and faster
 - Values passed in registers, rarely pushed on stack
 - Certain regs are volatile, others are non-volatile
- Be careful about data layout: 64-bit code may be different (pointers grow from 4 to 8 bytes)

Step 4: optimize

AMD64 Assembly Programmer's Model



Step 4: optimize

Assembly code example: software pipelining

asm code, calculates Mandelbrot set

```
movapd    xmm2, xmm0;
mulpd     xmm2, xmm1;    c = z_real x z_imag
mulpd     xmm0, xmm0;    a = z_real x z_real
mulpd     xmm1, xmm1;    b = z_imag x z_imag
subpd     xmm0, xmm1;    z_real = a - b
addpd     xmm2, xmm2;    c = c x 2
addpd     xmm0, xmm12;   z_real = z_real + c_real
movapd    xmm1, xmm2;
addpd     xmm1, xmm13;   z_imag = c + c_imag
```

Lots of data dependencies limiting performance...

what can we do to fix that?

Step 4: optimize

Use the extra registers: implement *software pipelining*

```
movapd    xmm2, xmm0;
movapd    xmm6, xmm4;
mulpd     xmm2, xmm1;    c = z_real x z_imag
mulpd     xmm6, xmm5;
mulpd     xmm0, xmm0;    a = z_real x z_real
mulpd     xmm4, xmm4;
mulpd     xmm1, xmm1;    b = z_imag x z_imag
mulpd     xmm5, xmm5;
subpd     xmm0, xmm1;    z_real = a - b
subpd     xmm4, xmm5;
addpd     xmm2, xmm2;    c = c x 2
addpd     xmm6, xmm6;
addpd     xmm0, xmm12;   z_real = z_real + c_real
addpd     xmm4, xmm14;
movapd    xmm1, xmm2;
movapd    xmm5, xmm6;
addpd     xmm1, xmm13;   z_imag = c + c_imag
addpd     xmm5, xmm15;
```

No need to overlap
the two chains
quite this tightly.

The CPU re-orders
instructions
aggressively, so
dependency chains
only need to be
reasonably close
together.


For max
performance...
experiment!

Working 2 independent data sets makes the code
run 35% faster here!

Step 5: beverage

- After you have your 64-bit code running like blazes...
- Enjoy a beverage of your choice

demo



32-bit and 64-bit
optimization and graphics demo

Demo summary

- Inserting a second dependency chain in the loop dramatically improves performance on both 32-bit and 64-bit
- 64-bit code benefits more, because of the extra SSE regs
- 64-bit mode provides a 30%+ boost over 32-bit mode

	<u>32-bit</u>	<u>64-bit</u>	
1 chain	.92 Gflop	1.01 Gflop	~10% 64-bit benefit
2 chains	<u>1.31 Gflop</u>	<u>1.72 Gflop</u>	~30% 64-bit benefit
Chain gain =	42%	70%	

Action items

Start now, on your 64-bit porting/optimization project

- Compile with /Wp64 all the time for both 32-bit and 64-bit, and use /O2b2 /GL /fp:fast for 64-bit, and use Profile Guided Optimization. See MSFT docs on PGO.
- Go to www.amd.com and get all the AMD docs
 - “Develop with AMD” and “AMD64 Developer Resource Kit”, Optimization Guide, Programmer’s Manuals, etc.
 - Download and use the **CodeAnalyst** profiler, for 32 and 64-bit code
 - Learn how to use the 64-bit PSDK compiler with VS 6 and .NET
 - Other presentations, including TechEd 2003 Barcelona: Win64 Porting
 - AMD Developer Center in Sunnyvale, CA! Visit us, or remote access

mike.wall@amd.com



More action items

- Go to Microsoft BetaPlace, get the OS and Platform SDK for “64-bit Extended Systems”, get VS2005 Whidbey beta
 - Windows 64-bit OS (1184) and Platform SDK (1173) on MS Betaplace
 - Current AMD64 Platform SDK based on VC6.0 libs; for ATL/MFC, CRT, STL 7.1 lib files: e-mail **libs7164@microsoft.com**
 - DirectX for AMD64: in the DirectX 9 beta builds (beta 3 SDK)
- Go to MSDN and Microsoft.com for more docs
 - Search for 64-bit, AMD64, or “64-bit Extended”
 - Read about new 64-bit compiler features, intrinsics, etc.
 - Especially read about “Whidbey” performance optimization features

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft and Windows are registered trademarks of Microsoft Corporation. Other product names used in this presentation are for identification purposes only and may be trademarks of their respective companies.



Community Resources

- Microsoft Community Resources
<http://www.microsoft.com/communities/default.mspx>
- Non-Microsoft Community Resources
<http://www.microsoft.com/communities/related/default.mspx>
- Newsgroups
Converse online with Microsoft Newsgroups, including Worldwide
<http://www.microsoft.com/communities/newsgroups/default.mspx>
- User Groups
Meet and learn with your peers
<http://www.microsoft.com/communities/usergroups/default.mspx>
- Attend a free chat
<http://www.microsoft.com/communities/chats/default.mspx>
- Attend a free web cast
<http://www.microsoft.com/usa/webcasts/default.asp>
- Most Valuable Professional (MVP)
<http://mvp.support.microsoft.com/>

evaluations

Don't forget to complete your
online Evaluation Form
and you could win
a MPx200
with Windows Mobile™ Software



Microsoft



Microsoft®